

On the DAG Decomposition

¹Yangjun Chen, ²Yibin Chen

University of Winnipeg

ABSTRACT

In this paper, we propose an efficient algorithm to decompose a directed acyclic graph G into a minimized set of node-disjoint chains, which cover all the nodes of G . For any two nodes u and v on a chain, if u is above v then there is a path from u to v in G . The best algorithm for this problem up to now needs $O(n^3)$ time, where n is the number of the nodes of G . Our algorithm, however, needs only $O(\kappa n^2)$ time, where κ is G 's width, defined to be the size of a largest node subset U of G such that for every pair of nodes $x, y \in U$, there does not exist a path from x to y or from y to x . More importantly, by the existing algorithm, $O(n^2)$ extra space (besides the space for G itself) is required to maintain the transitive closure of G to do the task while ours needs only $O(\kappa n)$ extra space.

Keywords: reachability queries, directed graphs, transitive closure, graph decomposition

The paper is a modification and extension of the results presented in 4th Intl. Conf. Big Data and Cloud Computing (BdCloud), 2014 IEEE [6].

1. INTRODUCTION

Let G be a directed acyclic graph (a DAG for short). A *chain cover* of G is a set C of *node-disjoint chains* such that it covers all the nodes of G , and for any two nodes u and v on a chain $p \in C$, if u is above v then there is a path from u to v in G . In this paper, we discuss an efficient algorithm to find a *minimized* C for G .

As an example, consider the DAG shown in Fig. 1(a). We can decompose it into a set of two chains, as shown in Fig. 1(b), which covers all the nodes of G . Fig. 1(c) shows another possible minimized decomposition.

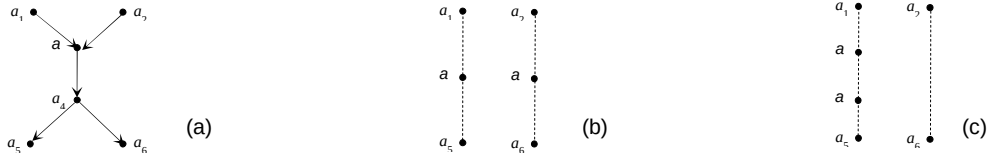


Fig. 1. Illustration for DAG decomposition

With the advent of the web technology, the efficient decomposition of a DAG G into a minimum set of chains becomes very important; especially, for the applications involving massive graphs such as social networks, for which we may quite often ask whether a node v is reachable from another node u through a path in G .

38 A naive method to answer such a query is to precompute the reachability between every pair
 39 of nodes in $G(V, E)$ - in other words, to compute the transitive closure of G , which is also a
 40 directed graph $G^*(V, E^*)$ with $(v, u) \in E^*$ if and only if there is a path from v to u in G . (See
 41 Fig. 2(a) for illustration, in which we show the transitive closure of the graph shown in Fig.
 42 1(a).)

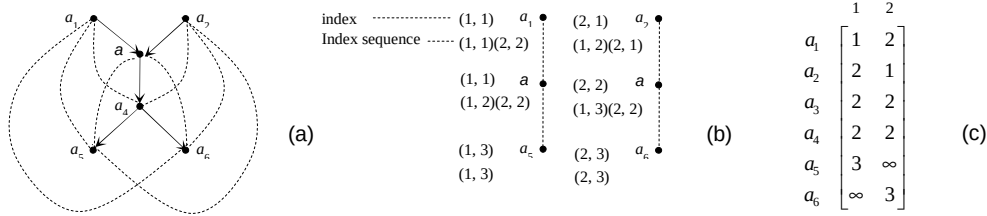


Fig. 2. Illustration for transitive closure and index sequences

43 As it is well known, the transitive closure of G can be stored as a boolean matrix M such that
 44 $M[i, j] = 1$ if there is path from i to j [21]; otherwise, $M[i, j] = 0$ [26]. Then, a reachability query
 45 can be answered in a constant time. However, this requires $O(n^2)$ space for storage, which
 46 makes it impractical for very large graphs, where $n = |V|$. Another method is to compute the
 47 shortest path from u to v over such a large graph on demand. Therefore, it needs only $O(m)$
 48 space, but with high query processing cost - $O(m)$ time in the worst case, where $m = |E|$.
 49 However, if we are able to decompose a DAG into a minimum set of chains, we can
 50 effectively compress a transitive closure without increasing much query time, as described
 51 below.

52 Let G be a directed graph. If it is cyclic (i.e., it contains cycles), we can first find all the
 53 *strongly connected components* (SCC) in linear time [25] and then collapse each of them
 54 into a representative node. Clearly, all of the nodes in an SCC are equivalent to its
 55 representative as far as reachability is concerned since each pair of nodes in an SCC are
 56 reachable from each other. In this way, we transform G to a DAG. Next, we decompose the
 57 DAG into a minimum set C of node-disjoint chains. (Recall that if a node u appears above
 58 another node v on a chain, there is a path from u to v .) Denote $|C| = \kappa$. We will then

- 61 (1) number each chain and number each node on a chain; and
- 62 (2) use a pair (i, j) as an index for the j th node on the i th chain.

63 Besides, each node u on a chain will be associated with an index sequence of the form: (r, j_r)
 64 $\dots (i, j_i) \dots (k, j_k)$ ($1 \leq r \leq i \leq k \leq \kappa$) such that any node v with index (x, y) is a descendant of u
 65 if and only if there exists (x, j_x) in the sequence with $y \geq j_x$. (See Fig. 2(b) for illustration.)
 66 Such index sequences can be created as follows.

67 First of all, we notice that we can associate each leaf node with an index sequence, which
 68 contains only one index, i.e., the index assigned to it. Clearly, such an index sequence is
 69 trivially sorted and its length is $1 \leq \kappa$. Let v be a non-leaf node with children $v_1, \dots, v_l$ each
 70 associated with an index sequence L_i ($1 \leq i \leq l$). Assume that $|L_i| \leq \kappa$ ($1 \leq i \leq l$) and the

indexes in each L_i are sorted according to the first element in each index. We will create an index sequence L for v , which initially contains only the index assigned to it. Then, we will merge all L_i 's into L one by one. To merge an L_i into L , we will scan both L and L_i from left to right. Let (a_1, b_1) (from L) and (a_2, b_2) (from L_i) be the index pairs currently encountered. We will perform the following checkings:

- If $a_2 > a_1$, we go to the index next to (a_1, b_1) (in L) and compare it with (a_2, b_2) in a next step.
- If $a_1 > a_2$, insert (a_2, b_2) just before (a_1, b_1) (in L). Go to the index next to (a_2, b_2) (in L_i) and compare it with (a_1, b_1) in a next step.
- If $a_1 = a_2$, we will compare b_1 and b_2 . If $b_1 < b_2$, nothing will be done. If $b_2 < b_1$, replace b_1 (in (a_1, b_1)) with b_2 . In both cases, we will go to the indexes next to (a_1, b_1) (in L) and (a_2, b_2) (in L_i), respectively.
- We will repeat the above three steps until either L or L_i is exhausted. If when L is exhausted L_i still has some remaining elements, append them at the end of L .

Obviously, after all L_i 's have been merged into L , the length of L is still bounded by the number κ . Denote by d_v the outdegree of v . The time spent on this process is then bounded by $O(\sum_v d_v \cdot \kappa) = O(\kappa m)$, but the space overhead is only $O(\kappa n)$. The query time remains $O(1)$

if we store the index sequences as a matrix $\mathbf{M}_G$, as shown in Fig. 2(c), in which each entry $\mathbf{M}_G(v, j)$ is the j th element in the index sequence associated with node v . So, a node u with index (i, j) is a descendant of node v if and only if $\mathbf{M}_G(v, i) \leq j$. In practice, κ is in general much smaller than n . In this sense, G^* is effectively compressed based on a minimized decomposition of G .

The problem to decompose a DAG is also heavily related to another theoretical problem [7, 8, 11, 19, 20]: the decomposition of *partially ordered sets* (or *posets* for short) $\mathbf{S} = (S, \succeq)$ into a minimum set of chains, where S is a set of elements and $\succeq$ is a *reflexive*, *transitive*, and *antisymmetric* relation over the elements. We can represent any poset $\mathbf{S}$ as a DAG G , where each node stands for an element in S and each arc $u \rightarrow v$ for a relation. Obviously, all the transitive relations in $\mathbf{S}$ can be represented by the transitive closure G^* of G . According to Dilworth [8], the size of a minimum decomposition equals the size of a maximum antichain U , which is a subset of elements such that for each two elements $a, c \in U$, $a \not\succeq c$ and $c \not\succeq a$. Furthermore, by using the Fulkerson's method [12], a minimum set of chains can be found in $O(n^3)$ time as follows:

- i) Construct the transitive closure G^* of G representing $\mathbf{S} = (S, \succeq)$.
- ii) Let $S = \{a_1, a_2, \dots, a_n\}$. Construct a bipartite graph G_S with bipartite (V_1, V_2) , where $V_1 = \{x_1, x_2, \dots, x_n\}$, $V_2 = \{y_1, y_2, \dots, y_n\}$ and an edge joins $x_i \in V_1$ to $y_j \in V_2$ whenever $a_i \rightarrow a_j \in G^*$.
- iii) Find a maximal matching M of G_S . Then, for any two edges $e_1, e_2 \in M$, if $e_1 = (x_i, y_k)$ and $e_2 = (x_k, y_j)$, connect e_1 to e_2 (by identifying y_k with x_k)

According to Fulkerson [12], the number of chains constructed as described above is $n - |M|$. It must be minimum since in terms of König's theorem ([2], page 180), the size of a maximum antichain U of $\mathbf{S}$ is also $n - |M|$ and we are not able to place any two elements in U on a same chain. Thus, we have $\kappa = |U|$, referred to as the width of G (or $\mathbf{S}$).

See Fig. 3 for illustration.

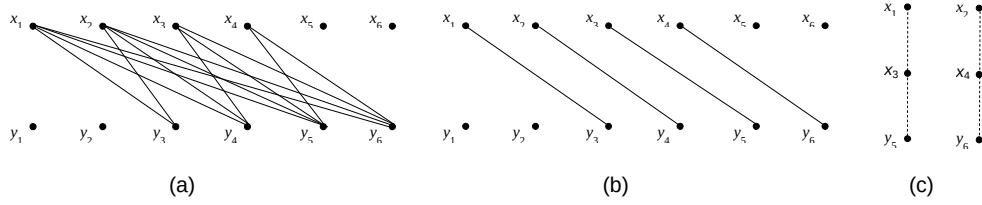


Fig. 3. Illustration for poset decomposition

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

The dominant cost of the above process is obviously the time and space for constructing G^* . They are bounded by $O(n^3)$ and $O(n^2)$, respectively [26]. However, using the algorithm proposed by Hopcroft and Karp [13], M can be found in $O(e \cdot \sqrt{n})$ time, where e is the number of the arcs in G^* , bounded by $O(n^2)$.

In [14], Jagadish discussed an algorithm for finding a minimum set of node-disjoint paths that cover a directed acyclic graph G by transforming the problem to a *min* network flow [9, 15, 18, 22]. Its time complexity is bounded by $O(n \cdot m)$. However, a chain is in general not a path. For any pair of nodes u and v on a chain, we only require that if u appears above v , there is a path from u to v . So, the number of paths found by the method discussed in [14] is generally much larger than the minimal number of node-disjoint chains. However, if we apply the Jagadish's method to G^* , we can get a minimized set of chains of G . But again, $O(n^3)$ time and $O(n^2)$ space are required to construct G^* .

The method discussed in [3] is also to decompose a DAG into node-disjoint chains. It runs in $O(n^{2.5})$ time. However, the decomposition found is not minimum. Our earlier algorithm [4] works for the same purpose. Its time complexity is bounded by $O(k^{1.5}n)$, where k is the number of the chains, into which the DAG is decomposed. But in some cases it fails to find a minimum set of chains since when generating chains, only part of reachability information is considered. This problem is removed by [5] and [6] both with the same time complexity $O(\kappa n^2)$. However, in the method discussed in [5] each node is associated with a large data structure and requires $O(\kappa n^2)$ space in the worst case. By [6], the generated chains may contain some newly created nodes, but how to remove such nodes are not discussed at all.

Different from the above strategies, the algorithm discussed in [10] is to find a maximum k -chain in a planar point set $M \subseteq N \times N$, where $N = \{0, 1, \dots, n-1\}$ and is defined by establishing (i', j') (i, j) if and only if $i' > i$ and $j' > j$. So M is a special kind of posets. A k -chain is a subset of M that can be covered by k chains. The time complexity of this algorithm is bounded by $O((n^2/k)/\log n)$. The algorithms discussed in [17] and [24] are to find a maximum 2-chain and 1-chain in M , respectively. [17] needs $\Theta(n \cdot \log n)$ time while [24] needs only $O(p \cdot n)$ time, where p is the length of the longest chain.

In this paper, we propose an efficient algorithm to find a minimum set of chains for G . It runs in $O(\kappa n^2)$ time and in $O(\kappa n)$ space while the best algorithm for this problem needs $O(n^3)$ time and in $O(n^2)$ space.

161 The remainder of the paper is organized as follows. In Section 2, we discuss an algorithm to
 162 stratify a DAG into different levels and review some concepts related to bipartite graphs, on
 163 which our method is based. Section 3 is devoted to the description of our algorithm to
 164 decompose a DAG into chains, as well as the analysis of its computational complexities. In
 165 Section 4, we prove the correctness of the algorithm. Finally, a short conclusion is set forth
 166 in Section 5.
 167

168 2. GRAPH STRATIFICATION AND BIPARTITE GRAPHS

169
 170 Our method is based on a DAG stratification strategy and an algorithm for finding a maximal
 171 matching in a bipartite graph. Therefore, the relevant concepts and techniques should be
 172 first reviewed and discussed.

173 174 2.1 Stratification of DAGs

175 We first discuss the DAG stratification.

176
 177 **Definition 1** Let $G(V, E)$ be a DAG. We decompose V into subsets $V_0, V_1, \dots, V_h$ such that V
 178 $= V_0 \cup V_1 \cup \dots \cup V_h$ and each node in V_i has its children appearing only in $V_{i-1}, \dots, V_0$ ($i = 1,$
 179 $\dots, h$), where h is the height of G , i.e., the length of the longest path in G . $\square$

180
 181 For each node v in V_i , we say, its level is i , denoted $level(v) = i$. We also use $C_j(v)$ ($j < i$) to
 182 represent a set of links which start from v to all those v 's children, which appear in V_j .
 183 Therefore, for each v in V_i , there exist $i_1, \dots, i_k$ ($i_l < i, l = 1, \dots, k$) such that the set of its
 184 children equals $C_{i_1}(v) \cup \dots \cup C_{i_k}(v)$. Let $V_i = \{v_1, v_2, \dots, v_{ij}\}$. We use $C_j^i(v_1)$ ($j < i$) to represent $C_j(v_1)$
 185 $\cup \dots \cup C_j(v_{ij})$.

186
 187 Such a DAG decomposition can be done in $O(m)$ time by using the following algorithm, in
 188 which we use $G_1 \setminus G_2$ to stand for a graph obtained by deleting the arcs of G_2 from G_1 ; and G_1
 189 $\cup G_2$ for a graph obtained by adding the arcs of G_1 and G_2 together. In addition, $d^-(v)$ and
 190 $d^+(v)$ represent v 's indegree and v 's outdegree, respectively.

191
 192 **Algorithm** *graph-stratification*(G)

193 **begin**

194 1. $V_0 :=$ all the nodes with no outgoing arcs; $i := 0$;
 195 2. $W :=$ all the nodes that have at least one child in V_0 ;
 196 3. **while** $W \neq \emptyset$ **do**
 197 4. { **for each** node v in W **do**
 198 5. { let $v_1, \dots, v_k$ be v 's children appearing in V_i ;
 199 6. $C_i(v) := \{v_1, \dots, v_k\}$; (*Here, for simplicity, we use v_j to represent a link from v to v_j .*)
 200 7. **if** $d^+(v) > k$ **then** remove v from W ;
 201 8. $G := G \setminus \{v \rightarrow v_1, \dots, v \rightarrow v_k\}$;
 202 9. $d^+(v) := d^+(v) - k$;
 203 10. }
 204 11. $V_{i+1} := W$; $i := i + 1$;
 205 12. $W :=$ all the nodes that have at least one child in V_i ;
 206 13. }
 207 **end**

208 In the above algorithm, we first determine V_0 , which contains all those nodes having no
 209 outgoing arcs (see line 1). In the subsequent computation, we determine $V_1, \dots, V_h$. In this
 210 process, G is reduced step by step (see line 8), so is $d^+(v)$ for any $v \in G$ (see line 9). In order
 211 to determine V_i ($i > 0$), we will first find all those nodes that have at least one child in V_{i-1} ,
 212 which are stored in a temporary variable W . For each node v in W (see line 3), we will then
 213 check whether it also has some other children not appearing in V_{i-1} , which is done by
 214 checking whether $d^+(v) > k$ in line 7, where k is the number of v 's children in V_{i-1} . If it is the
 215 case, it will be removed from W since it cannot belong to V_i . Concerning the correctness of
 216 the algorithm, we have the following proposition.

217
 218 **Proposition 1** Let $G_0 = G$. Denote by G_j the reduced graph after the j th iteration of the out-
 219 most **for**-loop. Denote by (v) the outdegree of v in G_j . Then, any node v in G_j does not have
 220 children appearing in $V_0 \cup \dots \cup V_{j-1}$, where V_0 contains all those nodes having no outgoing
 221 arcs, and for any $v \in V_i$ ($i = 1, \dots, j-1$) $(v) = 0$ while $(v) \neq 0, \dots, (v) \neq 0$.

222
 223 *Proof.* We prove the proposition by induction on j .

224
 225 Basic step. When $j = 1$, the proposition trivially holds.

226
 227 Induction hypothesis. Assume that when $j = l$, the proposition holds. Then, we have

228 (1) $G_l = G \setminus \left(\bigcup_{i=0}^{l-1} \bigcup_{v \in G} C_i(v) \right)$, and

229 (2) $d_l^+(v) = d^+(v) - \sum_{i=0}^{l-1} |C_i(v)|$.

230 Now we consider the case of $j = l + 1$. From lines 8 and 9, as well as the induction
 231 hypothesis, we immediately get

232 (3) $G_{l+1} = G \setminus \left(\bigcup_{i=0}^l \bigcup_{v \in G} C_i(v) \right)$, and

233 (4) $d_{l+1}^+(v) = d^+(v) - \sum_{i=0}^l |C_i(v)|$.

234 From (3) and (4), and also from lines 7 and 11, we can see that for any node $v \in V_{l+1}$ we
 235 have $d_{l+1}^+(v) = 0$ while $d_0^+(v) \neq 0, \dots, d_l^+(v) \neq 0$. $\square$

236
 237 From the proof of Proposition 1, we can see that

- 238
 239 • to check whether a node v in G_j belongs to V_{j+1} , we need only to check whether $d_l^+(v)$ is
 240 strictly larger than $|C_j(v)|$ (see line 7), which requires a constant time; and
 241 • G is correctly stratified.

242
 243 Since each arc is accessed only once in the process, the time complexity of the algorithm in
 244 bounded by $O(m)$.

245
 246 As an example, consider the graph shown in Fig. 4(a). Applying the above algorithm to this
 247 graph, we will generate a stratification of the nodes as shown in Fig. 4(b).

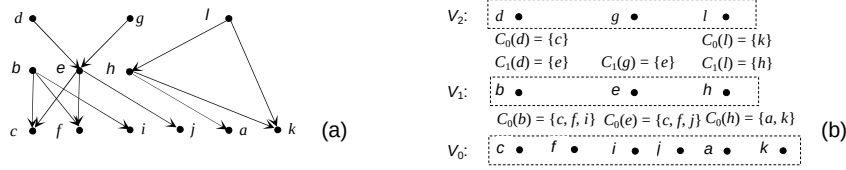


Fig. 4. Illustration for DAG stratification

In Fig. 4(b), the nodes of the DAG shown in Fig. 4(a) are divided into three levels: $V_0 = \{c, f, i, j, a, k\}$, $V_1 = \{b, e, h\}$, and $V_2 = \{d, g, l\}$. Associated with each node at each level is a set of links pointing to its children at different levels. For example, node b in V_1 is associated with three links respectively to nodes c , f , and i in V_0 , denoted as $C_0(b) = \{c, f, i\}$. (For simplicity, we use $C_0(b) = \{c, f, i\}$ to represent three links from b to c , f , and i , respectively.)

2.2 Concepts of bipartite graphs

Now we restate two concepts from the graph theory which will be used in the subsequent discussion.

Definition 2 (*bipartite graph* [2]) An undirected graph $B(V, E)$ is bipartite if the node set V can be partitioned into two sets T and S in such a way that no two nodes from the same set are adjacent. We also denote such a graph as $B(T, S; E)$. $\square$

For any node $v \in B$, $neighbor(v)$ represents a set containing all the nodes connected to v .

Definition 3 (*matching* [2]) Let $B(V, E)$ be a bipartite graph. A subset of edges $E' \subseteq E$ is called a *matching* if no two edges in E' have a common end node. A matching with the largest possible number of edges is called a *maximal matching*, denoted as M_B (or simply M if B is clear from context.) $\square$

Let M be a matching of a bipartite graph $B(T, S; E)$. A node v is said to be *covered* by M , if some edge of M is incident to v . We will also call an uncovered node *free*. A path or cycle is *alternating*, relative to M , if its edges are alternately in $E \setminus M$ and M . A path is an *augmenting path* if it is an alternating path with free origin and terminus. Let $v_1 - v_2 - \dots - v_k$ be an alternating path with $(v_i, v_{i+1}) \in E \setminus M$ and $(v_{i+1}, v_{i+2}) \in M$ ($i = 1, 3, \dots$). By transferring the edges on the path, we change it to another alternating path with $(v_i, v_{i+1}) \in M$ and $(v_{i+1}, v_{i+2}) \in E \setminus M$ ($i = 1, 3, \dots$). In addition, we will use $\phi_M(T)$ and $\phi_M(S)$ to represent all the free nodes in T and S , respectively; and use $\xi_M(T)$ for all the covered nodes in T and $\xi_M(S)$ for all the covered nodes in S . Finally, if $(u, v) \in M$, we say, u covers v with respect to M , and *vice versa*, denoted as $M(u) = v$, and $M(v) = u$, respectively.

Much research on finding a maximal matching in a bipartite graph has been done. The best algorithm for this task is due to Hopcroft and Karp [13] and runs in $O(m \cdot \sqrt{n})$ time, where $n = |V|$ and $m = |E|$. The algorithm proposed by Alt, Blum, Melhorn and Paul [1] needs $O(n^{1.5} \sqrt{m / (\log n)})$ time. In the case of large m , the latter is better than the former.

3. ALGORITHM DESCRIPTION

In this section, we describe our algorithm for the DAG decomposition. The main idea behind it is to construct a series of bipartite graphs for $G(V, E)$ based on the graph stratification and then find a maximum matching for each of such bipartite graphs using the Hopcroft-Karp algorithm [13]. All these matchings make up a set of node-disjoint chains, which, however, may not be minimal. In the following, we first discuss an example to illustrate this idea in 3.1. Then, in 3.2, we define the so-called *virtual nodes*, and show how they can be used to efficiently and effectively reduce the number of node-disjoint chains. Next, in 3.3, we discuss how the virtual nodes can be resolved (removed) from created chains to get the final result.

3.1 An example

Example 1 Consider the graph and the corresponding stratification shown in Fig. 4. A bipartite graph made up of V_0 and V_1 : $B(V_1, V_0; E_1)$ with $E_1 = E \cap (V_1 \times V_0)$ is shown in Fig. 5(a) and a possible maximal matching M_1 of it is shown in Fig. 5(b).

Another bipartite graph made up of V_1 and V_2 : $B(V_2, V_1; E_2)$ with $E_2 = E \cap (V_2 \times V_1)$ is shown in Fig. 5(c) and a possible maximal matching M_2 of it is shown in Fig. 5(d).

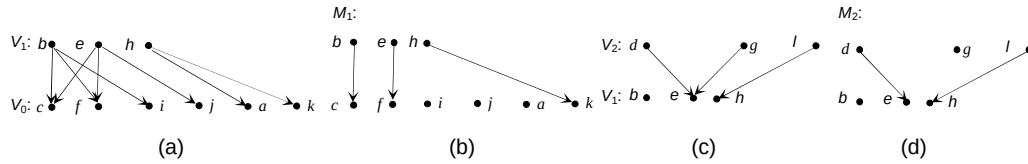


Fig. 5. Maximum matchings for bipartite graphs

Combining M_1 and M_2 by connecting their edges, we will get a set of seven chains, denoted by $M_1 \cup M_2$ and shown in Fig. 6(a). (Note that four of these chains each contain only a single node.)

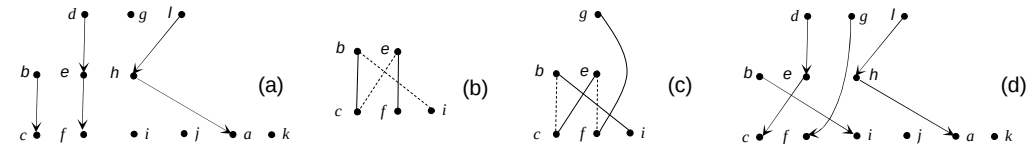


Fig. 6. Illustration for transferring edges on alternating paths

However, if we transfer the edges on an alternating path relative to M_1 : $f - e - c - b - i$ (see Fig. 6(b), where a solid edge represents an edge belonging to M_1 while a dashed edge to $E_1 \setminus M_1$); and connect g to f as illustrated in Fig. 6(c), we will get a set of six chains as shown in Fig. 6(d). (Note that g and f are on a chain since there exists a path $g - e - f$, which connects g and f .) $\square$

The question is how to efficiently find such a possible transformation to reduce the number of chains.

For this purpose, we introduce the concept of virtual nodes to transfer the reachability information and at the same time to maintain the information on how a transformation can be conducted.

3.2 Chain generation

From the above example, we can see that by simply combining maximal matchings of bipartite graphs, the number of formed chains may be larger than the minimized number of chains. To solve this problem, we need to introduce some virtual nodes into the original graph, which are used to transfer the reachability information from lower levels to higher levels.

3.2.1 Virtual nodes

We will work bottom-up. During the process, some virtual nodes may be added to V_i ($i = 1, \dots, h - 1$) level by level. However, such virtual nodes will be eventually resolved to obtain the final result.

In the following, we first give a formal definition of virtual nodes. Then, we describe how a virtual node is established. We start our discussion with the following specification:

$$V_0' = V_0.$$

$$V_i' = V_i \cup \{\text{virtual nodes added to } V_i\} \text{ for } 1 \leq i \leq h - 1.$$

$$\mathbf{C}_i = \mathbf{C}_{i-1}^i \cup \{\text{all the new arcs from the nodes in } V_i \text{ to the virtual nodes added to } V_{i-1}'\} \text{ for } 1 \leq i \leq h - 1.$$

$B(V_i, V_{i-1}'; \mathbf{C}_i)$ - the bipartite graph containing V_i and V_{i-1}' .

M_i - a maximal matching of $B(V_i, V_{i-1}'; \mathbf{C}_i)$.

Definition 4 (virtual nodes) Let $G(V, E)$ be a DAG, divided into $V_0, \dots, V_h$ (i.e., $V = V_0 \cup \dots \cup V_h$). Let M_i be a maximal matching of $B(V_i, V_{i-1}'; \mathbf{C}_i)$ for $i = 1, \dots, h$. For each free node v in V_{i-1}' with respect to M_i , a virtual node v' created for v is a new node added to V_i ($1 \leq i \leq h - 1$), denoted as $v = s(v')$. $\square$

The goal of virtual nodes is to establish the connection between the free nodes (with respect to a certain maximum matching of a bipartite graph) and the nodes that may be several levels apart. Therefore, for each virtual node v' (created for v in V_{i-1}' and added to V_i), a bunch of virtual arcs incident to it should be created. Especially, we distinguish among three kinds of virtual arcs: inherited arcs, transitive arcs and alternating arcs, which are created as follows.

inherited arcs - If there is $u \in V_j$ ($j > i$) such that $u \rightarrow v \in E$, add $u \rightarrow v'$, referred to as an inherited arc.

transitive arcs - If there exist $u \in V_j$ ($j > i$) and $w \in V_i$ such that $u \rightarrow w \in E$ and $w \rightarrow v \in \mathbf{C}_i$, add $u \rightarrow v'$ if it has not been created as an inherited arc, referred to as a transitive arc.

In order to create the *alternating arcs*, we need first to find a maximum set of node-disjoint alternating paths such that

1. each of them starts at a free node and ends at a covered node in V_{i-1}' ; and
2. they cover all the nodes in $B(V_i, V_{i-1}'; \mathbf{C}_i)$, except several (possibly 0) free nodes in V_{i-1}' and V_i .

For example, relative to the maximum matching shown in Fig. 5(b) for the bipartite graph shown in Fig. 5(a), we can find three such paths as shown in Fig. 7.

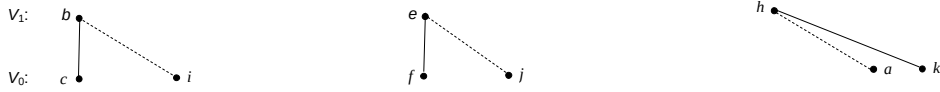


Fig. 7. A maximum set of alternating paths

369
370

371 Denote the path starting at free node v by P_v . The alternating arcs can be described as
372 below.

373 *alternating arc* - If there exists $w (\neq v)$ on P_v and $u \in V_j (j > i)$ such that one of the two
374 conditions holds:

- 375 - $u \rightarrow w \in E$, or
376 - there is a node $w' \in V_i$ such that $u \rightarrow w' \in E$ and $w' \rightarrow w \in C_i$,
377

378 add $u \rightarrow v'$ if it has not been created as an inherited or a transitive arc. It is referred to as an
379 alternating arc. We create such an arc to indicate a possibility to make v covered by
380 transferring the edges on the corresponding alternating path from v to w , and then connect u
381 and w (see Fig. 6(b) and (c) for illustration.)
382

383 In addition, a virtual arc from v' to $s(v')$ is generated to record the relationship between v'
384 and $s(v')$.
385

386 **Example 2** Continued with Example 1. Relative to M_1 of $B(V_1, V_0; E_1)$ shown in Fig. 5(b), i, j
387 and k are three free nodes. Then, three virtual nodes i', j' and k' (for i, j and k , respectively)
388 will be created and added to V_1 . Then, we have $V_1' = \{b, e, h, i', j', k'\}$. In addition, five virtual
389 arcs: $d \rightarrow i', d \rightarrow j', g \rightarrow i', g \rightarrow j'$, and $l \rightarrow k'$ will be generated, shown as five dashed arcs in
390 Fig. 8(a).
391

392 Among these virtual arcs, $l \rightarrow k'$ is an inherited arc since in the original graph we have $l \rightarrow k$
393 (see Fig. 4(a)). But $d \rightarrow j'$ and $g \rightarrow j'$ are two transitive arcs since j is reachable respectively
394 from d and g through e in V_1 (see Fig. 4(a)).
395

396 Finally, $d \rightarrow i'$ and $g \rightarrow i'$ are two alternating arcs. We join d and i' since there is a node f that
397 is connected to i through an alternating path: $f - e - c - b - i$ (see Fig. 6(b)) and f is
398 reachable from d through a node e in V_1 (see Fig. 4(a).) (We also note that c is another node
399 connected to i through an alternating path: $c - b - i$, and $d \rightarrow c \in E$. However, only one
400 alternating arc is created no matter how many possibilities we have to generate such an
401 arc.) For the same reason, we join g and i' .
402

403 In Fig. 8(b), we show a possible maximum matching M_2 of $B(V_2, V_1'; C_2)$. Combining M_2 and
404 M_1 , we get a set of six chains as shown in Fig. 8(c).
405

406 On these chains, the virtual nodes j' and k' can be simply removed since they do not have a
407 parent along the corresponding chains. In order to remove i' , however, we have to transfer
408 the edges on the alternating path: $f - e - c - b - i$ and then connect g and f , obtaining the
409 final chains shown in Fig. 6(d). We can also transfer the edges on $c - b - i$ and then connect
410 g and c to get a different set of six chains.
411
412

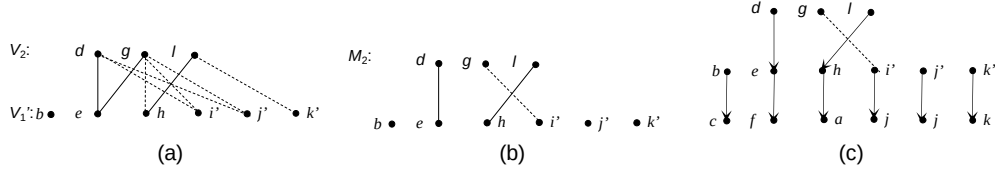


Fig. 8. Illustration for virtual nodes and chains

We will call an arc along a chain a *chain arc*. From the above example, we can see that how a virtual node is resolved depends on how it is connected to its parent through a chain arc. Especially, an alternating arc in fact does not represent a reachability, but indicates a possibility to connect two nodes by transferring edges along some alternating path. Thus, we need to label virtual arcs to represent their properties, and at the same time indicate at what level a virtual node is added. Let v' be a virtual node. Depending on whether its source $s(v')$ is an actual node or a virtual node itself, we label the virtual arcs incident to v' in two different ways.

Assume that $s(v')$ is an actual node in V_{i-1} . Then, v' is a virtual node added to V_i and a virtual arc incident to v' : $u \rightarrow v'$ with $u \in V_j$ ($j > i$) will be labeled as follows:

- i) If $u \rightarrow v'$ is inherited or transitive, its label $label(u \rightarrow v')$ will be set to 0, indicating that $s(v')$ is reachable from u (through a path).
- ii) If $u \rightarrow v'$ is an alternating arc, $label(u \rightarrow v')$ will be set to i , indicating that to resolve v' we need to transfer edges along an alternating path in $B(V_i, V_{i-1}; \mathbf{C}_i)$.

If $s(v')$ itself is a virtual node, we need to label $u \rightarrow v'$ a little bit differently:

- iii) If $u \rightarrow v'$ is inherited, the label for it is set to be the same as $label(u \rightarrow s(v'))$.
- iv) If $u \rightarrow v'$ is transitive, there must exist $w_1, \dots, w_k$ ($k \geq 1$) in V_i such that $w_1 \rightarrow s(v'), \dots, w_k \rightarrow s(v') \in \mathbf{C}_i$ and $u \rightarrow w_1, \dots, u \rightarrow w_k \in E$. We will label $u \rightarrow v'$ with $\min\{l_1, \dots, l_k\}$, where $l_j = label(w_j \rightarrow s(v'))$ ($j = 1, \dots, k$).
- v) If $u \rightarrow v'$ is an alternating arc, $label(u \rightarrow v')$ is set to i (in the same way as (ii)).

In addition, for convenience, all the original arcs in G are considered to be labeled with 0.

In the whole process, we will not only create a set of chains which may contain virtual nodes, but also a new graph by adding virtual nodes and virtual arcs to G , called a *companion graph* of G , denoted as G_c , which will be used for resolution of virtual nodes.

Example 3 Consider the graph shown in Fig. 9(a). This graph can be divided into four levels as shown in Fig. 9(b).

In Fig. 10(a), we show the bipartite graph $B(V_1, V_0; \mathbf{C}_1)$ made up of the first two levels. A possible maximal matching M_1 of it is shown in Fig. 10(b). Relative to M_1 , a, b and c are three free nodes in V_0 . So three virtual nodes a', b' and c' will be created and added to V_1 as shown in Fig. 10(c). At the same time, 13 arcs will be created. Among them, only one arc connects a node in V_3 to one of these virtual nodes while all the remaining 12 arcs connect some nodes in V_2 to them, as described below.

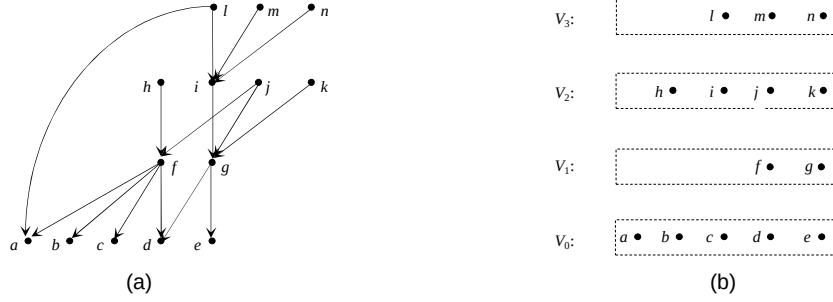


Fig. 9. A DGA and its stratification

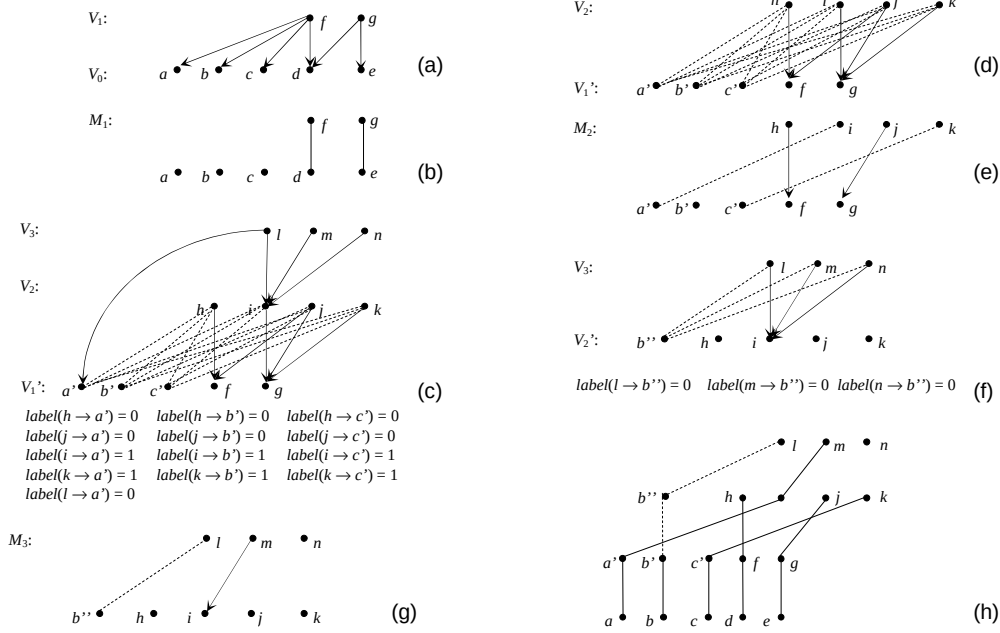


Fig. 10. Illustration for generation of paths

- The only new arc connecting a node l in V_3 to a virtual node a' : $l \rightarrow a'$ is an inherited arc, labeled with 0 according to (i). (There is neither alternating nor transitive arc connecting any node in V_3 to any of these virtual nodes.)
- The 12 new arcs connecting the nodes in V_2 to some virtual nodes can be divided into two groups.
 - Group 1 contains 6 transitive arcs, labeled with 0 according to (i):
 - $h \rightarrow a'$, $h \rightarrow b'$ and $h \rightarrow c'$.
 - $j \rightarrow a'$, $j \rightarrow b'$ and $j \rightarrow c'$.
 - Group 2 contains 6 alternating arcs, labeled with 1 according to (ii):
 - $i \rightarrow a'$, $i \rightarrow b'$ and $i \rightarrow c'$.
 - $k \rightarrow a'$, $k \rightarrow b'$ and $k \rightarrow c'$.

Thus, $V_1' = \{a', b', c', f, g\}$. $B(V_2, V_1'; \mathbf{C}_2)$ is shown in Fig. 10(d). Assume that the maximal matching M_2 found for it is as shown in Fig. 10(e). Relative to M_2 , b' is a free node in V_1' and then a virtual node b'' will be created and added to V_2 (see Fig. 10(f).) In addition, three transitive arcs: $l \rightarrow b''$, $m \rightarrow b''$ and $n \rightarrow b''$ will be created. All of them are labeled with 1.

$l \rightarrow b''$ is added because b' is reachable from l through i in V_1' . $label(l \rightarrow b'')$ is set to be 1 according to (iv) since $label(i \rightarrow b') = 1$. The same analysis applies to the other two arcs.

$B(V_3, V_2'; \mathbf{C}_3)$ is shown in Fig. 9(f). Fig. 10(g) shows a possible maximal matching M_3 of this bipartite graph. Combining M_1 , M_2 , and M_3 , we get $M_1 \cup M_2 \cup M_3$. This plus all the free nodes in V_3 make up a set of six chains as shown in Fig. 10(h), and one of them contains only a single node. This must be a minimum set since the original graph contains an antichain of 6 nodes: $\{a, b, c, k, m, n\}$.

In terms of the above discussion, we give the following algorithm.

Algorithm *GenChain(Stratification of G)*

input: a graph stratification.

output: a set of chains which may contain virtual nodes.

begin

1. $V_0' := V_0$;

2. **for** $i = 1$ to h **do**

3. { find M_i for $G(V_i, V_{i-1}'; \mathbf{C}_i)$

4. let $v_1, \dots, v_k$ be all the free nodes in V_{i-1}' , relative to M_i ;

5. $V_i' := V_i \cup \{v_1', \dots, v_k'\}$; ($v_1', \dots, v_k'$ are the virtual nodes created for $v_1, \dots, v_k$, respectively.)*

6. let $u_1, \dots, u_j$ be all the covered virtual nodes in V_{i-1}' ;

7. **for** $l = 1$ to j **do**

8. { remove all the virtual arcs incident to u_l , except the arc belonging to M_i ; }

9. **for** $l = 1$ to k **do**

10. { create all the virtual arcs incident to v_l' ; }

11. }

12. $M := M_1 \cup \dots \cup M_h$; return M ;

End

In the above algorithm, special attention should be paid to lines 6 - 8, by which all the virtual arcs incident to a covered virtual node in V_{i-1}' (i.e., a virtual node in V_{i-1}' , which is covered relative to M_i), except the corresponding arcs belonging to M_i , will be removed since they will not be used any more in the subsequent computation for the chain generation. Therefore, at any point in time, the number of virtual arcs maintained in the process is bounded by $O(k \cdot n)$.

3.2.2 Computational complexity of chain generation

We now analyze the time complexity of the chain generation.

In general, the cost of this process can be divided into two parts:

- cost₁: the time for finding a maximal matching of every $B(V_i, V_{i-1}'; \mathbf{C}_i)$ ($i = 1, \dots, h$; $V_0' = V_0$); and

517 - cost_2 : the time for generating virtual arcs.

518

519 We first prove three lemmas to show that for any $i \in \{1, \dots, h-1\}$ $|V'_i| \leq \kappa$.

520

521 Let M be a maximal matching of a bipartite graph $B(V_1, V_0; E)$. Let Y_i ($i = 0, 1$) be a subset of
522 V_i . We denote by $M(Y_i)$ ($i = 0, 1$) a subset of $V_{(i+1) \bmod 2}$ such that for each $v \in M(Y_i)$ there
523 exists a node u in Y_i with $(u, v) \in M$. We further divide M into three (possibly empty) groups:
524 $M[1]$, $M[2]$, and $M[3] = M \setminus (M[1] \cup M[2])$ such that

525

526 - for each $v \in \xi_{M[1]}(V_1)$, there exists at least one alternating path connecting a free node in
527 $\phi_M(V_0)$ to v ; (remember that $\phi_M(V_i)$ stands for all the free nodes in V_i relative to M while
528 $\xi_M(V_i)$ for all the covered nodes in V_i by M . So $\xi_{M[1]}(V_i)$ represents all those nodes in V_i
529 covered by $M[1]$.)

530 - for each $v \in \xi_{M[2]}(V_0)$, there exists at least one alternating path connecting a free node in
531 $\phi_M(V_1)$ to v ;

532 - for each $v \in \xi_{M[3]}(V_0)$, there exists no alternating path connecting it to any node in $\phi_M(V_1) \cup$
533 $\xi_{M[2]}(V_1)$; and for each $u \in \xi_{M[3]}(V_1)$, there exists no alternating path connecting it to any
534 node in $\phi_M(V_0) \cup \xi_{M[1]}(V_0)$.

535

536 Concerning this partition of M , we have the following lemma.

537

538 **Lemma 1** Any node in $\xi_{M[1]}(V_0)$ does not connect to any node in $\phi_M(V_1) \cup \xi_{M[2]}(V_1)$ through an
539 alternating path relative to M . Also, any node in $\xi_{M[2]}(V_1)$ does not connect to any node
540 in $\phi_M(V_0) \cup \xi_{M[1]}(V_0)$ through an alternating path relative to M .

541

542 *Proof.* Let v be a node in $\xi_{M[1]}(V_0)$. Assume that there is an alternating path P connecting v to
543 a free node u in $\phi_M(V_1)$. Then, the path from a free node w in $\phi_M(V_0)$ to $M(v)$, the edge $(M(v),$
544 $v)$, and P together make up an augmenting path connecting w and u , contradicting the fact
545 that M is a maximal matching. Therefore, any node in $\xi_{M[1]}(V_0)$ does not connect to any node
546 in $\phi_M(V_1)$. In the same way, we can prove the rest part of the lemma. $\square$

547

548 From this lemma, the following lemma can be immediately derived, by which we show how
549 to find an antichain for bipartite graph.

550

551 **Lemma 2** Let M be a maximal matching of a bipartite graph $B(V_1, V_0; E)$. Then, $\phi_M(V_0) \cup M$
552 $\cup \phi_M(V_1)$ make up a minimized set of chains of $B(V_1, V_0; E)$; and $\phi_M(V_0) \cup \phi_M(V_1) \cup \xi_{M[1]}(V_0)$
553 $\cup \xi_{M[2]}(V_1) \cup \xi_{M[3]}(V_0)$ is one of its antichains.

554

555 *Proof.* It is easy to see that $\phi_M(V_0) \cup M \cup \phi_M(V_1)$ is a minimized set of chains of $B(V_1, V_0; E)$.
556 It is also easy to see that any node in $\xi_{M[3]}(V_0)$ is not reachable from any node in $\phi_M(V_1)$
557 $\cup \xi_{M[2]}(V_1)$. Then, from Lemma 1, we can see that any node in $\xi_{M[1]}(V_0)$ is not reachable from
558 $\phi_M(V_1) \cup \xi_{M[2]}(V_1)$, and thus any pair of nodes in $A = \phi_M(V_0) \cup \phi_M(V_1) \cup \xi_{M[1]}(V_0) \cup \xi_{M[2]}(V_1) \cup$
559 $\xi_{M[3]}(V_0)$ are not reachable from each other. Therefore, A is an antichain. In addition, we have
560 $|\phi_M(V_0) \cup M \cup \phi_M(V_1)| = |A|$. So A is a maximum antichain. $\square$

561

562 **Lemma 3** Let $G(V, E)$ be a DAG, divided into $V_0, \dots, V_h$ (i.e., $V = V_0 \cup \dots \cup V_h$). Let M_i be a
563 maxima matching of the bipartite graph $B(V_i, V_{i-1}; \mathbf{C}_i)$. $V'_i = V_i \cup \{\text{virtual nodes added to } V_i\}$
564 for $1 \leq i \leq h-1$. Then, for any $i \in \{1, \dots, h-1\}$, we have $|V'_i| \leq \kappa$.

565 *Proof.* First, we notice that $|V_1'| = |V_1| + |\varphi_{M_1}(V_0)| = |\varphi_{M_2}(V_1) \cup \varphi_{M_1}(V_1) \cup \xi_{M_1[1]}(V_0)$
 566 $\cup \xi_{M_1[2]}(V_1) \cup \xi_{M_1[3]}(V_0)| \leq \kappa$. Then, we analyze the size of V_2' . Obviously, $|V_2'| = |V_2| +$
 567 $|\varphi_{M_2}(V_1')| = |A|$, where $A = \varphi_{M_2}(V_1') \cup \varphi_{M_2}(V_2) \cup \xi_{M_2[1]}(V_1') \cup \xi_{M_2[2]}(V_2) \cup \xi_{M_2[3]}(V_1')$.
 568 According to Lemma 2, A is an antichain of $G(V_2, V_1'; \mathbf{C}_2)$. Let $R = \varphi_{M_2}(V_1') \cup \xi_{M_2[1]}(V_1') \cup$
 569 $\xi_{M_2[3]}(V_1')$. We will distinguish between two cases:

570

571 1. R does not contain virtual nodes. Then, A is a set such that any two nodes in it are not
 572 connected through a path. Thus, $|A| \leq \kappa$

573 2. R contains at least a virtual node. In this case, we will replace each virtual node v in R
 574 with $s(v)$ (a free node in V_0) and each of those nodes u in R with $M_1(u)$ (a node in V_0 such
 575 that $(v, u) \in M_1$) which belong to $\xi_{M_1[1]}(V_1)$ and connected to a free node of V_0 relative to
 576 M_1 (through an alternating path), resulting in a new set R' with the following properties:

577

578 i) Any two nodes in R' are not connected through a path according to Lemma 1.

579 ii) Any node w in R' is not connected to a node in $D = \varphi_{M_2}(V_2) \cup \xi_{M_2[2]}(V_2)$. Otherwise, if
 580 w is a free node of V_0 relative to M_1 in R' , then its virtual node must be connected to
 581 that node in D . Contradiction. If w is a node in V_0 such that $w = M_1(u)$ for some u
 582 $\in \xi_{M_1[1]}(V_1)$, there must be a free node of V_0 relative to M_1 in R' , which is connected to a
 583 node in D . Contradiction.

584

585 Let $R'' = D \cup R'$. It can be seen that R'' is a set in which any two nodes are connected
 586 through a path. Therefore, $|A| = |R''| \leq \kappa$.

587

588 Repeating the above argument to all V_i' with $i \geq 3$, we can prove the lemma. $\square$

589

590 Based on Lemma 3, the time complexity of chain generation can be easily estimated.

591

592 First, using the Hopcroft and Karp algorithm [13], the time for finding a maximal matching of
 593 $B(V_i, V_{i-1}; \mathbf{C}_i)$ is bounded by

594

$$595 \quad O(\sqrt{|V_i| + |V_{i-1}'|} \cdot |\mathbf{C}_i|).$$

596

597 Therefore, cost_1 is bounded by

598

$$599 \quad O\left(\sum_{i=1}^k \left(\sqrt{|V_i| + |V_{i-1}'|} \cdot |\mathbf{C}_i|\right)\right)$$

$$600 \quad \leq O\left(\sqrt{k} \sum_{i=1}^k |V_i|\right) = O(\sqrt{\kappa} \cdot \kappa \cdot n)$$

601 For estimating cost_2 , we need to compute the costs for creating all the inherited, transitive
 602 and alternating arcs. First, for a virtual node, the cost for generating inherited arcs is a
 603 constant since we can simply promote the corresponding free node from its level to the level
 604 above it and handle it as virtual. (For example, to create a virtual node for a node v at level i ,
 605 we can add v to level $i + 1$. Then create a node containing only a link to v and leave it at
 606 level i , used as a representative of v .)

607

608 Secondly, the cost for creating the transitive arcs for all the virtual nodes added to V_i is
 609 obviously bounded by $O(|V_{i-1}'| \cdot |V_i| \cdot n)$. It is because at most $|V_{i-1}'|$ virtual nodes can be added

to V_i and the number of all the transitive arcs incident to each of these virtual nodes is bounded by $O(|V_i| \cdot n)$.

612

613 So the total cost for creating the transitive arcs is bounded by

614

$$\sum_{i=1}^h |V_{i-1}'| \cdot |V_i| \cdot n \leq \kappa \cdot n \cdot \sum_{i=1}^k |V_i| = O(\kappa n^2).$$

616

617 Finally, we notice that for generating the alternating arcs incident to the virtual nodes added
618 to V_i , we need to search $B(V_i, V_{i-1}'; \mathbf{C}_i)$ once for each of them, and the number of the arcs in
619 $B(V_i, V_{i-1}'; \mathbf{C}_i)$ is bounded by $O(|V_{i-1}'| \cdot |V_i|) \leq O(\kappa |V_i|)$. In addition, for each free node v (in V_{i-1}'),
620 the number of all those nodes z (in V_{i-1}') which are connected to v through an alternating
621 path is bounded by $|V_i|$ since each of such nodes must be covered relative to M_i . For each z ,
622 we need to search at most $|V_i|$ arcs to find all those w in V_i such that $w \rightarrow z \in \mathbf{C}_i$. Moreover,
623 for each w , we have to further find all those nodes u such that $u \rightarrow w \in E$. So the cost for
624 generating all the alternating arcs incident to v is bounded by $O(|V_{i-1}'| \cdot |V_i| + |V_i|^2 + |V_i| \cdot n)$.
625 Therefore, the total cost for creating all the alternating arcs is bounded by

626

$$\kappa \cdot \sum_{i=1}^h |V_{i-1}'| \cdot |V_i| + |V_i|^2 + |V_i| \cdot n = O(\kappa^2 \cdot n + \kappa n^2).$$

628

629 In terms of the above analysis, we have the following proposition.

630 **Proposition 2** The time for creating a minimized set of disjoint chains, which may contain
631 virtual nodes, is bounded by $O(\kappa^2 \cdot n + \kappa n^2)$.

632 The analysis of the space requirement of this process is quite simple. As mentioned in 3.2.1,
633 at any time point, we maintain at most $O(\kappa n)$ virtual arcs. Besides this, all the bipartite
634 graphs: $B(V_1, V_0; \mathbf{C}_1)$, $B(V_2, V_1'; \mathbf{C}_2)$, ..., $B(V_h, V_{h-1}'; \mathbf{C}_h)$ have to be maintained.

635

636 Thus, the space overhead is bounded by

637

$$O(\kappa n + \sum_{i=1}^h |V_{i-1}'| \cdot |V_i|) = O(\kappa n).$$

639

640 3.3 Virtual node resolution

641

642 After the chain generation, the next step is to resolve virtual nodes on chains. In the
643 following, we will first discuss the working process to remove virtual nodes in 3.3.1. Then, in
644 3.3.2, we analyze its computational complexities.

645

646 3.3.1 Removing virtual nodes

647 To remove virtual nodes from chains, we will work with the companion graph G_c . Two steps
648 will be carried out:

649 1. Connect some nodes according to the connectivity represented by the virtual nodes and
650 then remove them.

651 2. Establish new connections between free nodes by transferring edges along alternating
652 within a bipartite graph or cross more than one bipartite graph.

653 In the first step, we will check G_c top-down and remove virtual nodes level by level. Let v be
654 a virtual node in V'_i . Let $u_1, \dots, u_k$ be its parents in G_c . We will distinguish between two cases:
655 i) $s(v)$ is an actual node. If $\text{label}(u_j \rightarrow v) = 0$ ($j = 1, \dots, k$), connect u_j to $s(v)$. Otherwise,
656 $\text{label}(u_j \rightarrow v)$ must be equal to i , and we will connect u_j to each node w which is connected
657 to $s(v)$ through an alternating path in $B(V_i, V_{i-1}'; \mathbf{C}_i)$ and w is reachable from u_j .
658 ii) $s(v)$ itself is a virtual node. If $\text{label}(u_j \rightarrow v) > i$, connect u_j to $s(v)$ and $\text{label}(u_j \rightarrow s(v))$ is set
659 to be the same as $\text{label}(u_j \rightarrow v)$. Otherwise, $\text{label}(u_j \rightarrow v) = i$. In this case, we will connect
660 u_j to some nodes in the same way as (i).
661 See Fig. 11 for illustration.

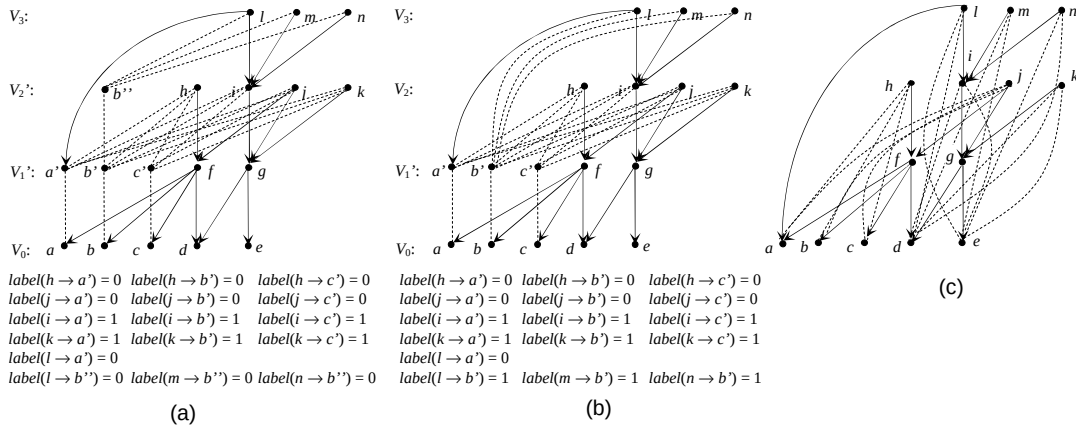


Fig. 11. Illustration for virtual node resolution

662

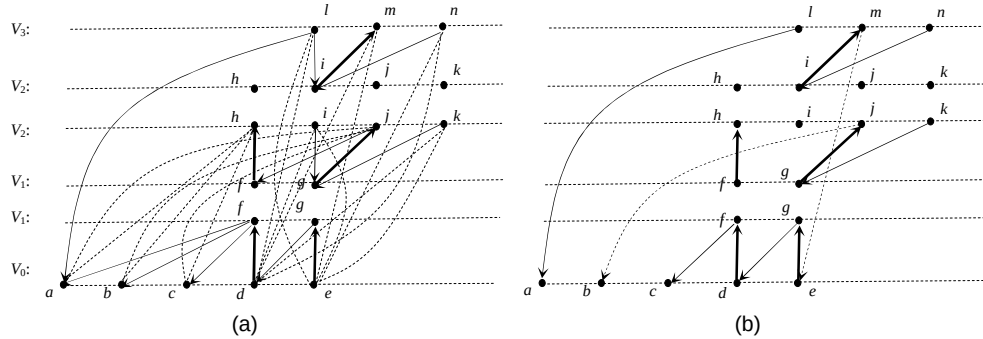


Fig. 12. Illustration for transferring edges along alternating paths

663

664 Let v be a virtual node and u be its parent along a chain. We call $u \rightarrow v$ is a chain arc. To
665 remove all the virtual nodes, we will go through three steps.

666

667 In the first step, all those virtual nodes that do not have a parent along a chain we will be
668 simply eliminated.

669

670 In the second step, we will check all those chain arcs with $label(u \rightarrow v) < level(v)$. For such
 671 an arc, we will directly connect u to $s(v)$ and then remove v . $label(u \rightarrow s(v))$ is set to be the
 672 same as $label(u \rightarrow v)$. If $label(u \rightarrow s(v)) < level(s(v))$, we will connect u to $s^2(v) = s(s(v))$ and
 673 remove $s(v)$. We repeat this operation until we meet $s^i(v)$ for some i such that $label(u \rightarrow s^i(v))$
 674 $= level(s^i(v))$.

675

676 After these two steps, we have only those virtual nodes v with $label(u \rightarrow v) = level(v)$ left on
 677 the chains, where u is the parent of v along the corresponding chain. Then, in the third step,
 678 we will remove all such virtual nodes by using alternating paths.

679

680 For this purpose, we first construct a combined graph, denoted as $\mathfrak{S}_G$, over all the bipartite
 681 graphs of G , based on G_c . We notice that in G_c all the generated virtual arcs are maintained.

682

683 1. A node in V_i ($i = 1, \dots, h-1$) will appear two times: one is in $\bar{B}(V_{i+1}, V'_i; \mathbf{C}_{i+1})$, and one is
 684 in $\bar{B}(V_i, V'_{i-1}; \mathbf{C}_i)$, but they are considered to be different nodes.

685

686 2. For each ending node of a chain $v \in V_i$ in $\bar{B}(V_i, V'_{i-1}; \mathbf{C}_i)$ ($i = 2, \dots, h$), we will search G
 687 starting from each node $u \in V_i$ which is connected to v through an alternating path in
 688 $\bar{B}(V_i, V'_{i-1}; \mathbf{C}_i)$, and connect u to each reachable node (in this way, part of the transitive
 689 closure is established.)

690

691 3. Remove all the remaining virtual nodes level by level. Let v be a virtual node in V'_i
 692 (in $\bar{B}(V_{i+1}, V'_i; \mathbf{C}_{i+1})$), connect each of its parent u to some nodes w in V_{i-1} (in $\bar{B}(V_i, V'_{i-1};$
 693 $\mathbf{C}_i)$) as below.

694

695 i) If $label(u \rightarrow v) = i$, w should be a node connected to $s(v)$ through an alternating path. w
 696 is actual node, and $label(u \rightarrow w)$ is implicitly set to be 0.

697

698 ii) If $label(u \rightarrow v) < i$, $w = s(v)$ and $label(u \rightarrow w)$ is set to be the same as $label(u \rightarrow v)$.

699

700 In Fig. 12, we show the construction of a combined graph over the bipartite graphs shown in
 701 Fig. 11.

702

703 Next, we need to define an important concept, the so-called alternating graphs.

704

705 **Definition 5** (alternating graph) Let $B(T, S; E)$ be a bipartite graph. Let M be a matching of
 706 $B(T, S; E)$. The alternating graph $\bar{B}$ with respect to M is a directed graph with the following
 707 sets of nodes and arcs:

708 $\bar{V} = V(\bar{B}) = T \cup S$, and

709 $\bar{E} = E(\bar{B}) = \{u \rightarrow v \mid u \in S, v \in T, \text{ and } (u, v) \in M\} \cup$
 710 $\{v \rightarrow u \mid u \in S, v \in T, \text{ and } (u, v) \in EM\}$. $\square$

711

712 By this definition, an alternating graph of a bipartite graph is almost the same as the original
 713 bipartite graph, but only with the covered and uncovered edges being set with reverse
 714 directions.

715

Now what we need to do is to find a maximum set of node-disjoint paths in the alternating version of the combined graph, each starting from an end node of a chain or a node which is a parent of some virtual node along a chain, and ending at a starting node of a chain.

This must be a minimum set since the original graph contains an antichain of size 6: $\{a, b, c, k, m, n\}$. $\square$

It remains to show how to find a maximal set of node-disjoint paths in $\mathfrak{S}_G$. For this purpose, we define a maximum flow problem over $\mathfrak{S}_G$ (with multiple sources and sinks) as follows:

- Each free node $v \in V_i$ in $\vec{B}(V_i, V_{i-1}; \vec{E}_i)$ (with respect to $M_i, i = 2, \dots, h$) is designated as a *source*. Each free node $u \in V_{j-1}$ in $\vec{B}(V_j, V_{j-1}; \vec{E}_j)$ (with respect to $M_j, j = 1, \dots, h-1$) is designated as a *sink*.
- Each arc $u \rightarrow v$ is associated with a capacity $c(u, v) = 1$. (If nodes u, v are not connected, $c(u, v)$ is considered to be 0.)

It is a typical 0-1 network. Finding a maximum flow corresponds to finding a maximum set of node-disjoint paths.

See Fig. 13(a), (b) and (c) for illustration. In Fig. 13(d), we show the final result obtained by transferring edges on the alternating paths.

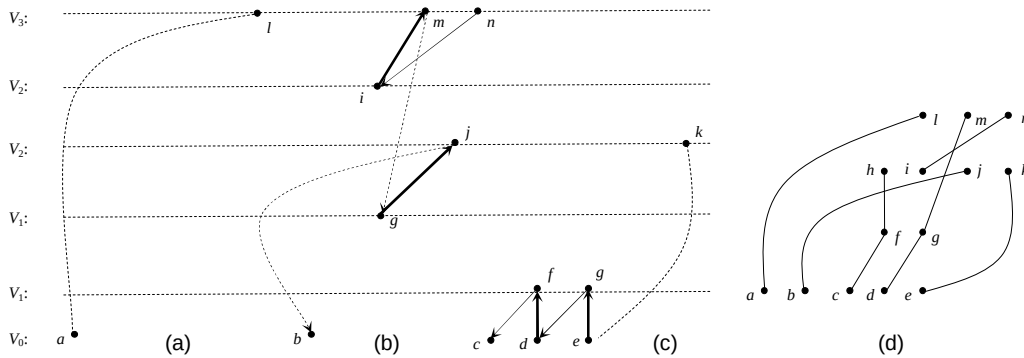


Fig. 14. A set of node-disjoint paths and the final result

3.3.2 Computational complexities of virtual node resolution

In this subsection, we analyze the time complexity of the virtual node resolution. The dominant cost of this process consists of two parts:

- cost_3 : the time for constructing combined graphs $\mathfrak{S}_G$; and
- cost_4 : the time for finding a maximum set of node-disjoint paths in $\mathfrak{S}_G$.

Let $u_1, \dots, u_k$ ($k < \kappa$) be all the end nodes of chains appearing in V_2 or higher. We will estimate the number n_i of the nodes connected to each u_i ($i = 1, \dots, k$) through an alternating path within the corresponding bipartite graph.

749 To this end, we consider the transitive closure $G^*(V, E^*)$ of G . We make a bipartite graph B_G
 750 as follows.
 751

- 752 1. For each $v \in V$, we produce two nodes x_v and y_v .
- 753 2. For any two nodes $u, v \in V$, we will connect x_u and y_v if $u \rightarrow v \in E^*$.

754 Let M be a maximum matching of B_G . We have $|M| \leq n$. Notice that the chains generated by
 755 Algorithm *GenChain(stratification of G)* corresponds a matching M' of B_G . Since on the
 756 chains almost each node is connected to another node and only some nodes are connected
 757 to virtual nodes, it is reasonable to assume that $|M'| = \alpha \cdot |M|$, where α is a constant with $0 < \alpha$
 758 ≤ 1 . The following proposition can be found in [2].
 759

760 **Proposition 3** $M \Delta M'$ contains at least $|M| - |M'|$ node-disjoint augmenting paths relative to
 761 M' , where $M \Delta M' = (M \cup M') \setminus (M \cap M')$, referred to as the symmetric difference of M and M' .
 762 $\square$
 763

764 From this proposition, the following lemma can be easily derived.
 765

766 **Lemma 4** The average length of an augmenting path relative to M' is $2 \lfloor |M'| / (|M| - |M'|) \rfloor + 1$.
 767 $\square$
 768

769 We have Lemma 4 since on average each augmenting path contains $\lfloor |M'| / (|M| - |M'|) \rfloor$
 770 edges from M' , and so $2 \lfloor |M'| / (|M| - |M'|) \rfloor + 1$ edges altogether.
 771

772 Since $|M'| = \alpha \cdot |M|$, we have
 773

$$774 \quad n_i = 2 \lfloor |M'| / (|M| - |M'|) \rfloor + 1 = 2 \lfloor \alpha / (1 - \alpha) \rfloor + 1.$$

775 In addition, each v_i ($i = 1, \dots, k$) can be at most on one of such node-disjoint augmenting
 776 paths, and k is bounded by κ . Therefore, the average value of cost_3 is bounded by
 777

$$778 \quad O\left(\sum_{i=1}^k n_i \cdot |E|\right) = O(\kappa m).$$

782 To estimate cost_4 , we need first to establish a lemma.
 783

784 **Lemma 5** For each arc e in $\mathfrak{S}_G$ corresponding to an edge in some maximum matching found
 785 for a bipartite graph, if is followed by another arc e' , then e' must be an arc which
 786 corresponds to an edge not belonging to any maximum matching. Similarly, any arc
 787 corresponding to an edge not covered by any maximum matching must be followed by an
 788 arc corresponding to a covered edge if any.
 789

790 *Proof.* Obviously, for any arc corresponding to an edge within a bipartite graph, the lemma
 791 holds. For any arc which crosses bipartite graphs, it always goes from some V_i in $\bar{B}(V_i, V_{i-1};$
 792 $\bar{E}_i)$ ($i = 2, \dots, h$) to a node in some V_j in $\bar{B}(V_{j+1}, V_j; \bar{E}_{j+1})$ with $j < i - 1$. So it cannot be an
 793 arc corresponding to a covered edge, but must be followed by an arc corresponding to a
 794 covered edge. $\square$
 795
 796

797 From this lemma, the following corollary can be immediately derived.

798

799 **Corollary 1** For each node v in $\mathfrak{S}_G$, either there is only one arc emanating from it or only one
800 arc entering it. $\square$

801

802 According to Corollary 1, we know that by using Dinic's algorithm [9] the time required to find
803 a maximum flow and then a maximum set of node-disjoint paths is bounded by
804 $O(|V(\mathfrak{S}_G)|^{0.5} \cdot |E(\mathfrak{S}_G)|)$ (see pp. 119 – 121 in [2] for a detailed analysis.) For self-explanation,
805 we also give a modified version of Dinic's algorithm adapted to our problem in Appendix.

806

807 **Proposition 4** The time complexity of the whole process to resolve virtual nodes is bounded
808 by $O(\kappa n^2 + \kappa m)$.

809

810 4. CORRECTNESS

811

812 In this subsection, we prove the correctness of our algorithm.

813

814 First, for any graph which can be divided into two levels the algorithm obviously produce a
815 correct answer.

816

817 For any graph which can divided into three levels, the result created by the algorithm is also
818 correct, as stated in the following lemma.

819

820 **Lemma 6** For any three-level graph $G(V, E)$, the number of chains generated by the
821 algorithm is minimum.

822

823 *Proof.* Let $V = V_0 \cup V_1 \cup V_2$. If no virtual node is created, the lemma obviously holds. So we
824 assume that some virtual nodes for the free nodes in V_0 relative to M_1 . In this case, all the
825 virtual nodes v will be removed by constructing $\mathfrak{S}_G$, in which each node on an alternating
826 path starting from a free node in V_2 relative to M_2 will be connected to all reachable nodes in
827 V_0 . Also, v 's parent (in V_2) will be connected to all the reachable nodes in V_0 . By transferring
828 edges on each path of a maximum set of node-disjoint paths each starting from a parent of a
829 virtual node or a free node in V_2 , and terminating at a free node in V_1 or V_0 , we will get the
830 result. It must be a minimum set of chains since the number of the free nodes which become
831 covered is maximized. $\square$

832

833 **Proposition 5** The number of the chains generated for a DGA by our algorithm is minimum.

834

835 *Proof.* We prove the proposition by induction on h .

836

837 Initial step. When $h = 1, 2$, the proposition holds according to Lemma 6.

838

839 Induction step. Assume that for any DAG of height k , the proposition holds. Now we consider
840 the case when $h = k + 1$. First, we construct a new graph G' from $G(V, E)$ as follows:

841

842 1. Stratify G , dividing V into $V_0, \dots, V_h$ (i.e., $V = V_0 \cup \dots \cup V_h$).

843

844 2. Find a maximum matching M_1 of $G(V_1, V_0; \mathbf{C}_1)$. Construct virtual nodes for all the free
845 nodes in V_0 , and add them into V_1 . Add all the virtual arcs as described in 3.2. Then,
846 remove V_0 .

847

848 So G' is of height k . According to the induction hypothesis, a minimal set Q of disjoint chains
849 can be found.

850

851 Let $u_1, \dots, u_k$ ($k < \kappa$) be all the end nodes of chains appearing in V_2 or higher. We connect
 852 each node on an alternating path starting from a u_i ($1 \leq i \leq k$) to a reachable node in V_0 ; and
 853 then connect the parents of the virtual nodes to all the respective reachable nodes in V_0 .
 854 Removing the virtual nodes. In a next step, we will find a maximum set of node-disjoint paths
 855 each starting from a u_i or a parent of some virtual node along a chain, and ending at a free
 856 node in V_0 . Assume that there are still l free nodes not on any of such node-disjoint paths.
 857 Thus, Q and these l free nodes make up a set of chain. It must be minimum since Q is
 858 minimum and l is also minimum. It should be the same as the result by applying the
 859 algorithm to G up to the edge transferring, as illustrated in Fig. 14. $\square$
 860

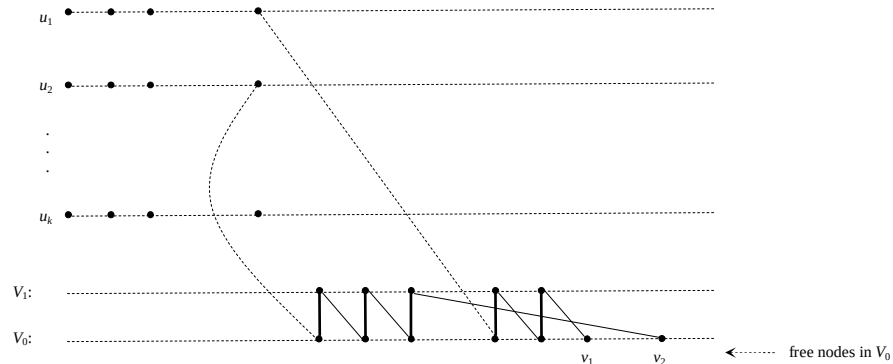


Fig. 14. A set of node-disjoint paths; each of them is an alternating path

861

862 5. CONCLUSION

863 In this paper, a new algorithm for finding a minimal decomposition of DAGs is proposed. The
 864 algorithm needs $O(\kappa \cdot n^2)$ time and $O(\kappa \cdot n + m)$ space, where n and m are the number of the
 865 nodes and the arcs in a DAG G , respectively; and κ is the width of G . The main idea of the
 866 algorithm is the concept of virtual nodes and the DAG stratification that generates a series of
 867 bipartite graphs which may contain virtual nodes. By executing Hopcroft-Karp's algorithm,
 868 we find a maximum matching for each of such bipartite graphs, which make up a set of
 869 node-disjoint chains. A next step is needed to resolve all the virtual nodes appearing on the
 870 chains to get the final result.

871

872 We also point out that our algorithm can be easily modified to a 0-1 network flow algorithm
 873 by defining a chain to be a path and accordingly changing the conditions for creating
 874 transitive and alternating arcs.

875

876 References

- 877 1. H. Alt, N. Blum, K. Mehlhorn, and M. Paul, Computing a maximum cardinality matching in
 878 a bipartite graph in time $O()$, *Information Processing Letters*, 37(1991), 237 -240.
- 879 2. A.S. Asratian, T. Denley, and R. Hagkvist, *Bipartite Graphs and their Applications*,
 880 Cambridge University, 1998.
- 881 3. C. Chekuri and M. Bender, An Efficient Approximation Algorithm for Minimizing Makespan
 882 on Uniformly Related Machines, *Journal of Algorithms* 41, 212-224(2001).
- 883 4. Y. Chen and Y.B. Chen, An Efficient Algorithm for Answering Graph Reachability Queries,
 884 in *Proc. 24th Int. Conf. on Data Engineering (ICDE 2008)*, IEEE, April 2008, pp. 892-901.

- 885 5. Y. Chen and Y.B. Chen, On the Decomposition of Posets, in *Proc. 2nd Int. Conf. on*
 886 *Computer Science and Service System (CSSS 2012)*, IEEE, Aug. 11-13, Nanjing, China, pp.
 887 1115 - 1119.
- 888 6. Y. Chen and Y.B. Chen, On the Graph Decomposition, *Int. Conf. on Big Data and Cloud*
 889 *Computing*, IEEE, Dec. 3 - 5, 2014, Sydney, Australia, pp. 777-784.
- 890 7. G.B. Dantzig and A. Hoffman, On a theorem of Dilworth, *Linear Inequalities and related*
 891 *systems* (H.W. Kuhn and A.W. Tucker, eds.) Annals of Math. Studies 38(1966), 207-214.
- 892 8. R.P. Dilworth, A decomposition theorem for partially ordered sets, *Ann. Math.* **51** (1950),
 893 pp. 161-166.
- 894 9. E.A. Dinic, Algorithm for solution of a problem of maximum flow in a network with power
 895 estimation, *Soviet Mathematics Doklady*, 11(5):1277-1280, 1970.
- 896 10. S. Felsner, L. Wernisch, Maximum k -chains in planar point sets: combinatorial structure
 897 and algorithms, *SIAM J. Comp.* **28**, 1998, pp. 192-209.
- 898 11. T. Gallai and A.N. Milgram, Verallgemeinerung eines Graphentheoretischen Satzes von
 899 Reedei. *Acta Sci. Math. Hung.*, 21(1960), 429-440.
- 900 12. D.R. Fulkerson, Note on Dilworth's embedding theorem for partially ordered sets, *Proc.*
 901 *Amer. Math. Soc.* 7(1956), 701-702.
- 902 13. J.E. Hopcroft, and R.M. Karp, An $n^{2.5}$ algorithm for maximum matching in bipartite
 903 graphs, *SIAM J. Comput.* 2(1973), 225-231.
- 904 14. H.V. Jagadish, "A Compression Technique to Materialize Transitive Closure," *ACM*
 905 *Trans. Database Systems*, Vol. 15, No. 4, 1990, pp. 558 - 598.
- 906 15. A.V. Karzanov, Determining the Maximal Flow in a Network by the Method of Preflow,
 907 *Soviet Math. Dokl.*, Vol. 15, 1974, pp. 434-437.
- 908 16. E.L. Lawler, *Combinatorial Optimization and Matroids*, Holt, Rinehart, and Winston, New
 909 York (1976).
- 910 17. R.-D. Lou, M. Sarrafzadeh, An optimal algorithm for the maximum two-chain problem,
 911 *SIAM J. Disc. Math.* **5**(2), 1992, pp. 285-304.
- 912 18. V.M. Malhotra, M.P. Kumar, and S.N. Maheshwari, An $O(|V|^3)$ Algorithm For Finding
 913 Maximum Flows in Networks, Computer Science Program, Indian Institute of Technology,
 914 Kanpur 208016, India, 1978.
- 915 19. M.A. Perles, A proof of Dilworth's decomposition theorem for partially ordered sets, *Israel*
 916 *J. of Math.* 1(1963), 105-107.
- 917 20. H. Tverberg, On Dilworth's decomposition theorem for partially ordered sets, *J. Comb.*
 918 *Th.* 3(1967), 305-306.
- 919 21. D. Coppersmith, and S. Winograd. Matrix multiplication via arithmetic progression.
 920 *Journal of Symbolic Computation*, vol. 9, pp. 251-280, 1990.
- 921 22. S. Even, *Graph Algorithms*, Computer Science Press, Inc., Rockville, Maryland, 1979.
- 922 23. L. Lamport, Time, clocks, and the ordering of events in a distributed system,
 923 *Communication of the ACM* 21(7), July 1978, 95-114.
- 924 24. H. Goeman, Time and Space Efficient Algorithms for Decomposing Certain Partially
 925 Ordered Sets, PhD thesis, Department of Mathematics-Science, Rheinischen Friedrich-
 926 Wilhelms Universität Bonn, Germany, Dec. 1999.
- 927 25. R. Tarjan: Depth-first Search and Linear Graph Algorithms, *SIAM J. Comput.* Vol. 1. No.
 928 2. June 1972, pp. 146 -140.

929 26. H.S. Warren, "A Modification of Warshall's Algorithm for the Transitive Closure of Binary
 930 Relations," *Commun. ACM* 18, 4 (April 1975), 218 - 220.
 931

932 APPENDIX

933 Find node-disjoint paths in Combined graphs

934

935 A.1 Algorithm

936

937 In the appendix, we discuss an algorithm for finding a maximal set of node-disjoint paths in a
 938 combined graph $\mathfrak{S}_G$. Its time complexity is bounded by $O(e \cdot \sqrt{n})$, where $n = V(\mathfrak{S}_G)$ and $e =$
 939 $E(\mathfrak{S}_G)$. It is in fact a modified version of Dinic's algorithm [6], adapted to combined graphs, in
 940 which each path from a virtual node to a free node relative to a maximum matching M_i for
 941 some bipartite graph is an alternating path, and for each edge $(u, v) \in M_i$, we have $d^+(u) =$
 942 $d^-(v) = 1$. Therefore, for any three nodes v, v' , and v'' on a path in $\mathfrak{S}_G$, we have $d^+(v) = d^-(v')$
 943 $= 1$, or $d^+(v) = d^-(v'') = 1$. We call this property the *alternating property*, which enables us to
 944 do the task efficiently by using a dynamical arc-marking mechanism. An arc $u \rightarrow v$ with $d^+(u)$
 945 $= d^-(v) = 1$ is called a *bridge*.
 946

947 Our algorithm works in multiple phases. In each phase, the arcs in $\mathfrak{S}_G$ will be marked or
 948 unmarked. We also call a virtual node in $\mathfrak{S}_G$ an origin and a free node a terminus. An origin
 949 is said to be saturated if one of its outgoing arcs is marked; and a terminus is saturated if
 950 one of its incoming arcs is marked.
 951

952 In the following discussion, we denote $\mathfrak{S}_G$ by A .
 953

954 At the very beginning of the first phase, all the arcs in A are unmarked.
 955

956 In the k th phase ($k \geq 1$), a subgraph $A^{(k)}$ of A will be explored, which is defined as follows.
 957

958 Let V_0 be the set of all the unsaturated origins. Define V_j ($j > 0$) (note that not to confuse this
 959 with the graph stratification) as follows:
 960

$$\begin{aligned} 961 \quad E_{j-1} &= \{u \rightarrow v \in E(A) \mid u \in V_{j-1}, v \notin V_0 \cup V_1 \cup \dots \cup V_{j-1}, u \rightarrow v \text{ is unmarked}\} \cup \\ 962 \quad &\quad v \rightarrow u \in E(A) \mid u \in V_{j-1}, v \notin V_0 \cup V_1 \cup \dots \cup V_{j-1}, v \rightarrow u \text{ is marked}\}, \\ 963 \quad V_j &= \{v \in V(A) \mid \text{for some } u, u \rightarrow v \text{ is unmarked and } u \rightarrow v \in E_{j-1}\} \cup \\ 964 \quad &\quad v \in V(A) \mid \text{for some } u, v \rightarrow u \text{ is marked and } v \rightarrow u \in E_{j-1}\}. \end{aligned}$$

965
 966 Define $j^* = \min\{j \mid V_j \cap \{\text{unsaturated terminus}\} \neq \emptyset\}$.
 967

968 $A^{(k)}$ is formed with $V(A^{(k)})$ and $E(A^{(k)})$ defined below.
 969

970 If $j^* = 1$, then

$$\begin{aligned} 971 \quad V(A^{(k)}) &= V_0 \cup (V_{j^*} \cap \{\text{unsaturated terminus}\}), \\ 972 \quad E(A^{(k)}) &= \{u \rightarrow v \mid u \in V_{j^*-1}, \text{ and } v \in \{\text{unsaturated terminus}\}\}. \end{aligned}$$

973
 974 If $j^* > 1$, then

$$\begin{aligned} 975 \quad V(A^{(k)}) &= V_0 \cup V_1 \cup \dots \cup V_{j^*-1} \cup (V_{j^*} \cap \{\text{unsaturated terminus}\}), \\ 976 \quad E(A^{(k)}) &= E_0 \cup E_1 \cup \dots \cup E_{j^*-2} \cup \{u \rightarrow v \mid u \in E_{j^*-1}, \text{ and } v \in \{\text{unsaturated terminus}\}\}. \end{aligned}$$

977 The sets V_j are called *levels*.

978

979 In $A^{(k)}$, a node sequence $v_1, \dots, v_j, v_{j+1}, \dots, v_l$ is called a complete sequence if the following
980 conditions are satisfied.

981

982 (1) v_1 is an origin and v_l is a terminus.

983 (2) For each two consecutive nodes v_j, v_{j+1} ($j = 1, \dots, l - 1$), we have an unmarked arc $v_j \rightarrow v_{j+1}$
984 in $A^{(k)}$, or a marked arc $v_{j+1} \rightarrow v_j$ in $A^{(k)}$.

985

986 Our algorithm will explore $A^{(k)}$ to find a set of node-disjoint complete sequences (i.e., no two
987 of them share any nodes.) Then, we mark and unmark the arcs along each complete
988 sequence as follows.

989

990 (i) If (v_j, v_{j+1}) corresponds to an arc in $A^{(k)}$, mark that arc.

991 (ii) If (v_{j+1}, v_j) corresponds to an arc in $A^{(k)}$, unmark that arc.

992

993 Obviously, if for an $A^{(k)}$ there exists j such that $V_j = \emptyset$ and $V_i \cap \{\text{unsaturated terminus}\} = \emptyset$
994 for $i < j$, we cannot find a complete sequence in it. In this case, we set $A^{(k)}$ to $\emptyset$ and then the
995 k th phase is the last phase.

996

997 **Example 4** Consider a graph A shown in Fig. 15(a), in which nodes a and b are two origins;
998 and nodes g and h are two terminus. Initially, all the arcs are not marked. Thus, $V_0 = \{a, b\}$,
999 $V_1 = \{c, d\}$, $V_2 = \{e, f\}$, $V_3 = \{g, h\}$; and $j^* = 3$. $A^{(1)}$ is the same as A . (Normally, $A^{(1)}$ has fewer
1000 nodes than A .)

1001

1002 Assume that by exploring $A^{(1)}$ (using the algorithm given below), we find a complete
1003 sequence: a, d, f, g . Then, we will mark three arcs (a, d) , (d, f) , and (f, g) , as shown by the
1004 thick arrows in Fig. 15(b). With respect to these marked arcs, a second subgraph $A^{(2)}$ (in the
1005 second phase) will be constructed as shown in Fig. 15(c). In this phase, $V_0 = \{b\}$ (since node
1006 a is saturated), $V_1 = \{d\}$, $V_2 = \{a\}$ (note that $a \rightarrow d$ is marked), $V_3 = \{c\}$, $V_4 = \{e\}$, $V_5 = \{g\}$, $V_6 =$
1007 $\{f\}$ (note that $f \rightarrow g$ is marked), $V_7 = \{h\}$; and $j^* = 7$. By exploring $A^{(2)}$, we will find another
1008 complete sequence: b, d, a, c, e, g, f, h , on which all the unmarked arcs will be marked while
1009 all the marked arcs will be unmarked, as demonstrated in Fig. 15(d). Fig. 16(e) shows all the
1010 marked arcs, which make up two node-disjoint paths: $a \rightarrow c \rightarrow e \rightarrow g$ and $b \rightarrow d \rightarrow f \rightarrow h$.

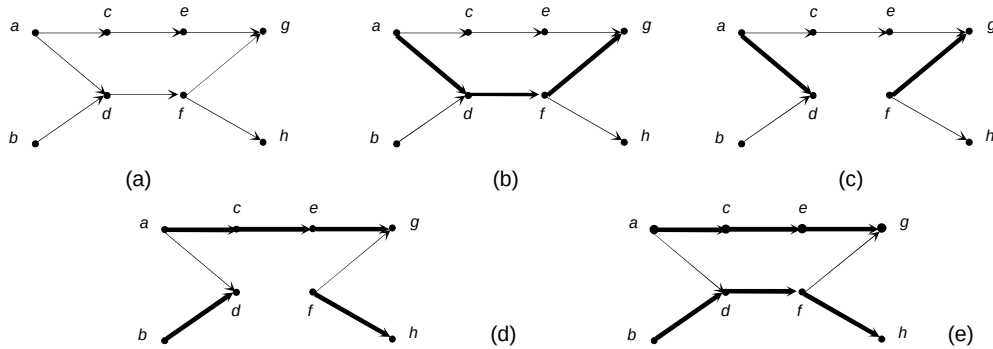


Fig. 15. Illustration for graph searching

1011

1012

1013 The following algorithm is devised to explore an $A^{(k)}$, in which a stack H is used to store
1014 complete sequences. In addition, for each v in $A^{(k)}$, $neighbor(v)$ represents a set of nodes: $v_1,$

1015 ..., v_m such that for each $j \in \{1, \dots, m\}$ $v \rightarrow v_j \in E(A^{(k)})$ if $v \rightarrow v_j$ is unmarked, or $v_j \rightarrow v \in E(A^{(k)})$
 1016 if $v_j \rightarrow v$ is marked.

1017

1018 **Algorithm** *subgraph-exploring()*

1019 **begin**

1020 1. let v be the first element in V_0 ;
 1021 2. *push*(v, H); mark v 'accessed';
 1022 3. **while** H is not empty **do** {
 1023 4. $v := \text{top}(H)$; (*the top element of H is assigned to v .*)
 1024 5. **while** $\text{neighbor}(v) \neq \emptyset$ **do** {
 1025 6. let u be the first element in $\text{neighbor}(v)$;
 1026 7. **if** u is accessed **then** remove u from $\text{neighbor}(v)$
 1027 8. **else** {*push*(u, H); mark u 'accessed'; $v := u$;}
 1028 9. }
 1029 10. **if** v is neither in V_j^* nor in V_0 **then** *pop*(H)
 1030 11. **else** {**if** v is in V_j^* **then** output all the elements in H ;
 1031 (*all the elements in H make up a complete sequence.*)
 1032 remove all elements in H ;
 1033 let v be the next element in V_0 ;
 1034 14. *push*(v, H); mark v ;
 1035 15. }

1036 **End**

1037

1038 The above algorithm works top-down, searching level by level. In each iteration of the outer
 1039 **while**-loop, a complete sequence is explored (by executing the inner **while**-loop, lines 5 - 9)
 1040 and stored in the stack H . All the found complete sequences are node-disjoint since any
 1041 repeated access of a node is blocked by using the mark 'accessed' (see lines 2, 7, and 8.)
 1042 Based on the above algorithm, the whole process to find a maximal set of node-disjoint
 1043 paths is given below.

1044

1045 **Algorithm** *node-disjoint-paths(A)*

1046 **begin**

1047 1. $k := 1$;
 1048 2. construct $A^{(1)}$;
 1049 3. **while** $A^{(k)} \neq \emptyset$ **do** {
 1050 4. call *subgraph-exploring*($A^{(k)}$);
 1051 5. let $P_1, \dots, P_l$ be all the found complete sequences;
 1052 6. **for** $j = 1$ to l **do**
 1053 7. { let $P_j = v_1, v_2, \dots, v_m$;
 1054 8. mark $v_i \rightarrow v_{i+1}$ or unmark $v_{i+1} \rightarrow v_i$ ($i = 1, \dots, m - 1$) according to (i) and (ii) above;
 1055 9. }
 1056 10. $k := k + 1$; construct $A^{(k)}$;
 1057 11. }

1058 **end**

1059

1060 The above algorithm runs in several phases. In each phase, a $A^{(k)}$ is constructed (see line 2,
 1061 and 10). Then, *subgraph-exploring()* is invoked to find all node-disjoint complete sequences.
 1062 Also, the arcs along each complete sequence will be marked or unmarked (see line 8).
 1063 When we meet a $A^{(k)} = \emptyset$, the algorithm terminates; and all the marked arcs in A make up a
 1064 maximal set of node-disjoint paths.

1065 **A.2 Time complexity and correctness**

1066 In this subsection, we analyze the time complexity of *node-disjoint-paths*(A) and prove its
 1067 correctness.

1068 **A.2.1 Time complexity**

1069 **Lemma 7** Let $V_1^{(k)}, \dots, V_{j_k}^{(k)}$ be the levels constructed when establishing $A^{(k)}$. Then, we have

1070 $j_k < j_{k+1}$ if $(k + 1)^{\text{th}}$ phase is not the last.

1071

1072 *Proof.* Let v be a node in $V_{j_k}^{(k)}$. If $v \notin V_{j_k}^{(k)}$, then, according to the definition of j^* , we must go
 1073 along a longer path from a node in $V_1^{(k+1)}$ to v than any path from a node in $V_1^{(k)}$ to v . If $v \in$
 1074 $V_{j_k}^{(k)}$, then, to reach v , we must go along a node sequence $v_1, \dots, v_{j_{k+1}} = v$ such that there
 1075 exist at least two consecutive nodes v_l, v_{l+1} ($1 < l < j_{k+1}$) with $v_{l+1} \rightarrow v_l$ being marked in the k^{th}
 1076 phase due to the alternating property. Going from v_l to v_{l+1} means a detour around. In terms
 1077 of the definition of j^* , $j_{k+1} > j_k$. $\square$

1078

1079 **Lemma 8** Let P be a maximal set of node-disjoint paths in A . Let $V_1^{(k)}, \dots, V_{j_1}^{(k)}$ be the levels
 1080 when establishing $A^{(1)}$. Then, $j_1 \leq |V(A)|/|P|$.

1081

1082 *Proof.* Let α be the length of the shortest path in P . Then, we have

1083

$$\alpha \cdot |P| \leq |V(A)|.$$

1084

1085 Therefore, $\alpha \leq |V(A)|/|P|$. However, $j_1 \leq \alpha$. Thus, the lemma follows. $\square$

1086

1087 For each $A^{(k)}$, we define $B^{(k)}$ as follows.

1088

1089 (1) If $u \rightarrow v \in E(A^{(k)})$ is an unmarked arc, then $u \rightarrow v \in E(B^{(k)})$.

1090

1091 (2) If $u \rightarrow v \in E(A^{(k)})$ is a marked arc, then $v \rightarrow u \in E(B^{(k)})$.

1092

1093 We will prove that $B^{(k)}$ also has the alternating property.

1094

1095 **Lemma 9** In $B^{(k)}$, for each node v , we have either $d^+(v) \leq 1$ or $d^-(v) \leq 1$.

1096

1097 *Proof.* We prove the lemma by induction on k .

1098

1099 Basis step. When $k = 1$. The lemma trivially holds.

1100

1101 Induction step. Assume that for $k \leq h$, the lemma holds. We consider $B^{(h+1)}$. Let v be a node
 1102 in $B^{(h+1)}$. If v does not appear in any complete sequences found in $A^{(h)}$. The indegree and
 1103 outdegree of v are the same as in $B^{(h)}$. If v appears in a complete sequence found in $A^{(h)}$, we do
 1104 the following analysis. Let $v_1, \dots, v_{i-1}, v, v_{i+1}, \dots, v_l$ be a complete sequence found in $A^{(h)}$, on
 1105 which v appears. Consider v_{i-1}, v, v_{i+1} , we have $d^+(v) = d^-(v_{i+1}) = 1$, or $d^+(v_{i-1}) = d^-(v) = 1$

1106 according to the induction hypothesis. In both cases, neither of $v_{i-1} \rightarrow v$ and $v \rightarrow v_{i+1}$ appear
 1107 in $B^{(h+1)}$. But in the former case, $u \rightarrow v$ (for some u) and $v \rightarrow v_{i-1}$ will be added into $B^{(h+1)}$, as
 1108 shown by the dashed arrows in Fig. 16(a). Note that in $B^{(h+1)}$ $d^+(v) = d^+(v_{i-1}) = 1$. In the latter
 1109 case, $v_{i+1} \rightarrow v$ and $v \rightarrow u'$ (for some u'). In this case, $d^+(v_{i+1}) = d^+(v_i) = 1$. See Fig. 16(b) for
 1110 illustration. Thus, a bridge in is replaced with a different bridge in $B^{(h+1)}$. $\square$



Fig. 16. Illustration for Lemma 9

1111
1112

1113 **Proposition 6** The time complexity of the algorithm *node-disjoint-paths*(A) is bounded by
 1114 $O(\sqrt{|V(A)|} |E(A)|)$.

1115

1116 *Proof.* Let P be a maximal set of node-disjoint paths in A . If $|P| \leq \sqrt{|V(A)|}$, then the number
 1117 of phases is bounded by $\sqrt{|V(A)|}$, and the result follows. If $|P| > \sqrt{|V(A)|}$, we consider k
 1118 such that in the k^{th} phase the number of node-disjoint paths is $|P| - \sqrt{|V(A)|}$. Then, by
 1119 constructing $A^{(k+1)}, \dots, A^{(m)}$ (assume that the m^{th} phase is the last), and then exploring them,
 1120 we will find the rest node-disjoint paths. Assume that $V_1^{(k+1)}, \dots, V_{j_{k+1}}^{(k+1)}$ be the levels
 1121 constructed when establishing $A^{(k+1)}$. In terms of Lemma 7 and 8, $j_{k+1} \leq |V(A)| / \sqrt{|V(A)|} =$
 1122 $\sqrt{|V(A)|}$.

1123

1124 In terms of Lemma 7, k must be less than j_{k+1} . Therefore, $k \leq \sqrt{|V(A)|}$. Thus, the time spent
 1125 for the first k phase is bounded by $O(\sqrt{|V(A)|} |E(A)|)$. Also, the time for finding the rest
 1126 node-disjoint paths is bounded by $O(\sqrt{|V(A)|} |E(A)|)$. So the total cost is $O(\sqrt{|V(A)|} |E(A)|)$.

1127 $\square$

1128

1129 A.2.2 Correctness

1130 **Lemma 10** Let $A^{(0)} = A, A^{(1)}, \dots, A^{(k)} = \emptyset$ be the graphs generated in the different phases
 1131 during the execution of Algorithm *node-disjoint-paths*(A). Then, for each marked arc $u \rightarrow v$ in
 1132 $A^{(i)}$ ($i = 1, \dots, k - 1$), the following conditions are satisfied.

1133

- 1134 i) If u is not an origin, then there exists a node u' such that $u' \rightarrow u$ is marked in $A^{(i)}$.
- 1135 ii) If v is not a terminus, then there exists a node v' such that $v' \rightarrow v$ is marked in $A^{(i)}$.

1136

1137 *Proof.* We prove condition (1) by induction on phases i .

1138

1139 Basic step. When $i = 1$. The proof is trivial.

1140

1141 Induction step. Assume that the lemma holds for $i \leq j$. Consider phase $j + 1$. Let $v_1, \dots, v_l$ be a
 1142 complete sequence found in $A^{(j+1)}$. Without loss of generosity, assume that (v_g, v_{g+1}) ($1 \leq g <$

1143 $l)$ corresponds to a marked arc. If (v_{g-1}, v_g) corresponds to a marked arc, condition (i) is
 1144 proved. Otherwise, there exists a subsequence $v_h, v_{h+1}, \dots, v_g$ such that each pair (v_{r+1}, v_r)
 1145 corresponds to a marked arc in $A^{(j)}$. According to the induction hypothesis, if v_g is not an
 1146 origin, there must exist a node v such that (v, v_g) corresponds to a marked arc in $A^{(j)}$.
 1147 According to the algorithm, (v_{r+1}, v_r) will be unmarked, but (v, v_g) remains marked.

1148

1149 In the same way, we can prove condition (ii). $\square$

1150

1151 From the proof of Lemma 10, we can also see that any two paths made up of marked arcs
 1152 (from an origin to a terminus) are node-disjoint.

1153

1154 **Proposition 7** The number of the node-disjoint paths in A found by *node-disjoint-paths*(A) is
 1155 maximum.

1156

1157 *Proof.* Let $A^{(0)} = A, A^{(1)}, \dots, A^{(k)} = \emptyset$ be the graphs generated in the different phases during
 1158 the execution of the algorithm. Let $V_0, V_1, \dots, V_j$ be the levels when creating $A^{(k)}$. Then, $V_j = \emptyset$
 1159 and $V_i \cap \{\text{unsaturated terminus}\} = \emptyset$ for $i < j$. Consider the cut $(R, \bar{R})$, where $R = V_0 \cup V_1 \cup$
 1160 $\dots \cup V_{j-1}$ and $\bar{R} = V(A) \setminus (V_0 \cup V_1 \cup \dots \cup V_{j-1})$. Each arc $u \rightarrow v$ with $u \in R$ and $v \in \bar{R}$ must be
 1161 marked. Otherwise, $V_j \neq \emptyset$. In addition, each two of such arcs are not on a same path.
 1162 According to Lemma 10, each of such arcs corresponds to a node-disjoint path and the
 1163 number of such arcs is maximum. This completes the proof. $\square$